Lord Of The Ring0 - Part 3 | Sailing to the land of the user (and debugging the ship)

idov31.github.io/2022/10/30/lord-of-the-ring0-p3

October 30, 2022

Prologue

In the last blog post, we understood what it is a callback routine, how to get basic information from user mode and for the finale created a driver that can block access to a certain process. In this blog, we will dive into two of the most important things there are when it comes to driver development: How to debug correctly, how to create good user-mode communication and what lessons I learned during the development of Nidhogg so far.

This time, there will be no hands-on code writing but something more important - how to solve and understand the problems that pop up when you develop kernel drivers.

Debugging

The way I see it, there are 3 approaches when it comes to debugging a kernel: The good, the great and the hacky (of course you can combine them all and any of them). I'll start by explaining every one of them, the benefits and the downsides.

- The good: This method is for anyone because it doesn't require many resources and is very effective. All you need to do is to set the VM where you test your driver to produce a crash dump (you can leave the crash dump option to automatic) and make sure that in the settings the disable automatic deletion of memory dumps when the disk is low is checked or you can find yourself very confused to not find the crash dump when it should be generated. Then, all you have to do is to drag the crash dump back to your computer and analyze it. The con of this method is that sometimes you can see corrupted data and values that you don't know how they got there, but most of the time you will get a lot of information that can be very helpful to trace back the source of the problem.

- The great: This method is for those who have a good computer setup because not everyone can run it smoothly, to debug your VM I recommend following these instructions. Then, all you have to do is put breakpoints in the right spots and do the debugging we all love to hate but gives the best results as you can track everything and see everything in real-time. The con of this method is that it requires a lot of resources from the computer and not everyone (me included) has enough resources to open Visual Studio, run a VM and remote debug it with WinDBG.

- The hacky: I highly recommend not using this method alone. Like in every type of program you can print debugging messages with KdPrint and set up the VM to enable debugging messages and fire up DbgView to see your messages. Make sure that if you are printing a string value lower the IRQL like so:

```
KIRQL prevIrql = KeGetCurrentIrql();
KeLowerIrql(PASSIVE_LEVEL);
KdPrint(("Print your string %ws.\n", myString));
KeRaiseIrql(prevIrql, &prevIrql);
```

Because it lets you see what the values of the current variables are it is very useful, just not if you did something that causes the machine to crash, that's why I recommend combining it with either the crash dump option or the debugging option.

I won't do here a guide on how to use WinDBG because there are many great guides out there but I will add a word about it. The top commands that help me a lot during the process of understanding what's wrong are:

- `!analyze -v` : It lets WinDBG load the symbols, what is the error code and most importantly the line in your source code that led to that BSOD.
- `lm` : This command shows you all the loaded modules at the time of the crash and allows you to iterate them, their functions, etc.
- `uf /D <address>` : This command shows you the disassembly of a specific address, so you can examine it.

After we now know the basics of how to debug a driver, let's dive into the main event: how to properly exchange data with the user mode.

Talking with the user-mode 102

Last time we understood the different methods to send and get data from user mode, the basic usage of IOCTLs and what IRPs are. But what happens when we want to send a list of different variables? What happens if we want to send a file name, process name or something that isn't just a number?

DISCLAIMER: As I said before, in this series I'll be using the IOCTL method, so we will address the problem using this method.

To properly send data we can use the handly and trusty struct. What you need to do is to define a data structure in both your user application and the kernel application for what you are planning to send, for example:

```
struct MyItem {
    int type;
    int price;
    WCHAR* ItemsName;
}
```

And send it through the DeviceIoControl:

```
DeviceIoControl(hFile, IOCTL_DEMO,
        &myItem, sizeof(myItem),
        &myItem, sizeof(myItem), &returned, nullptr)
```

But all of this we knew before, so what is new? As you noticed, I sent myItem twice and the reason is in the definition of DeviceIoControl:

```
BOOL DeviceIoControl(
  [in]                HANDLE       hDevice,
  [in]                DWORD        dwIoControlCode,
  [in, optional]      LPVOID       lpInBuffer,
  [in]                DWORD        nInBufferSize,
  [out, optional]     LPVOID       lpOutBuffer,
  [in]                DWORD        nOutBufferSize,
  [out, optional]     LPDWORD      lpBytesReturned,
  [in, out, optional] LPOVERLAPPED lpOverlapped
);
```

We can define the IOCTL in a way that will allow the driver to both receive data and send data, all we have to do is to define our IOCTL with the method type METHOD_BUFFERED like so:

```
#define IOCTL_DEMO CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

And now, SystemBuffer is accessible for both writing and reading.

A quick reminder: SystemBuffer is the way we can access the user data, and is accessible to us through the IRP like so:

```
Irp->AssociatedIrp.SystemBuffer;
```

Now that we can access it there several questions remain: How can I write data to it without causing BSOD? And how can I verify that I get the type that I want? What if I want to send or receive a list of items and not just one?

The second question is easy to answer and is already shown up in the previous blog post:

```
auto size = stack->Parameters.DeviceIoControl.InputBufferLength;

if (size % sizeof(MyItem) != 0) {
    status = STATUS_INVALID_BUFFER_SIZE;
    break;
}
```

This is a simple yet effective test but isn't enough, that is why we also need to verify every value we want to use:

```
...
auto data = (MyItem*)Irp->AssociatedIrp.SystemBuffer;

if (data->type < 0 || !data->ItemsName || data->price < 0) {
    status = STATUS_INVALID_PARAMETER;
    break;
}
...
```

This is just an example of checks that need to be done when accessing user mode data, and everything that comes or returns to the user should be taken care of with extreme caution.

Writing data back to the user is fairly easy like in user mode, the hard part comes when you want to return a list of items but don't want to create an entirely new structure just for it. Microsoft themselves solved this in a pretty strange-looking yet effective way, you can see it in several WinAPIs for example when iterating a process or modules and there are two approaches:

The first one will be sending each item separately and when the list ends send null. The second method is sending first the number of items you are going to send and then sending them one by one. I prefer the second method (and you can also see it implemented in Nidhogg) but you can do whatever works for you.

Conclusion

This time, it was a relatively short blog post but very important for anyone that wants to write a kernel mode driver correctly and learn to solve their problems.

In this blog, we learned how to debug a kernel driver and how to properly exchange data between our kernel driver to the user mode. In the next blog, we will understand the power of callbacks and learn about the different types that are available to us.

I hope that you enjoyed the blog and I'm available on Twitter, Telegram and by Mail to hear what you think about it! This blog series is following my learning curve of kernel mode development and if you like this blog post you can check out Nidhogg on GitHub.